Filas

Uma fila é uma estrutura de dados que admite remoção de elementos e inserção de novos elementos. Mais especificamente, uma *fila* (= *queue*) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção,

o elemento removido é o que está na estrutura há mais tempo.

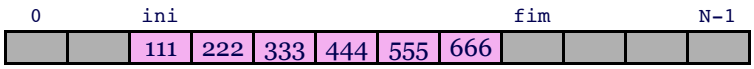
Em outras palavras, o primeiro objeto inserido na fila é também o primeiro a ser removido. Essa política é conhecida pela sigla FIFO (= *First-In-First-Out*).

Implementação em um vetor

Suponha que nossa fila mora em um vetor `fila[0..N-1]`. (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, caracteres, ponteiros, etc.) Digamos que a parte do vetor ocupada pela fila é

`fila[ini..fim-1]`.

O primeiro elemento da fila está na posição `ini` e o último na posição `fim-1`. A fila está *vazia* se `ini == fim` e *cheia* se `fim == N`. A figura mostra uma fila que contém os números 111, 222, ... , 666:



Para *remover*, ou *retirar* (= *delete* = *de-queue*), um elemento da fila basta fazer

```
x = fila[ini++];
```

Isso equivale ao par de instruções "`x = fila[ini]; ini += 1;`", nesta ordem. É claro que você só deve fazer isso se tiver certeza de que a fila não está vazia.

Para *colocar*, ou *inserir* (= *insert* = *enqueue*), um objeto `y` na fila basta fazer

```
fila[fim++] = y;
```

Note como esse código funciona bem mesmo quando a fila está vazia. É claro que você só deve inserir um elemento na fila se ela não estiver cheia; caso contrário, a fila *transborda* (ou seja, ocorre um *overflow*). Em geral, a tentativa de inserir em uma fila cheia é uma situação excepcional, que indica um mau planejamento lógico do seu programa.

Exercícios 1

1. Escreva uma função que devolva o comprimento (ou seja, o número de elementos) de uma fila dada.
2. Escreva funções `sai` e `entra` para remover e inserir em uma fila. Lembre-se de que uma fila é um pacote com

dois objetos: um vetor e dois índice. Quais os parâmetros de suas funções? Não use variáveis globais.

3. Suponha que, diferentemente da convenção adotada acima, a parte do vetor ocupada pela fila é `fila[ini..fim]`. Escreva a instrução que retira um elemento da fila. Escreva a instrução que coloca um objeto `y` na fila.

4. Suponha que, diferentemente da convenção adotada acima, a fila ocupa sempre a parte `fila[0..fim-1]` do vetor `fila`. Escreva código para remover um elemento da fila. Escreva código para inserir um novo objeto `y` na fila.

5. NADA A VER COM FILAS. Reescreva a seguinte linha de código com layout decente:

```
for(i = 23; i --> 0;)
```

Escreva uma versão mais simples da linha.

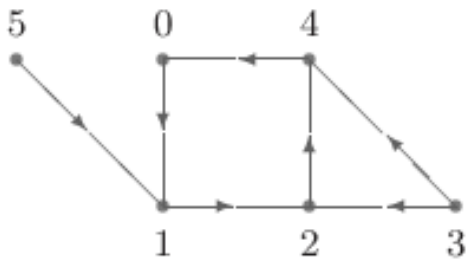
Aplicação: distâncias

O cálculo de distâncias em um grafo é uma aplicação clássica do conceito de fila. Imagine 6 cidades numeradas de 0 a 5 e interligadas por estradas de mão única. (É claro que você pode trocar "6" pelo seu número favorito.) As ligações entre as cidades são representadas por uma matriz `A` da seguinte maneira:

`A[i][j]` vale `1` se existe estrada de `i` para `j`

e vale 0 em caso contrário. Suponha que a matriz tem zeros na diagonal, embora isso seja irrelevante. Exemplo:

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0



A *distância* de uma cidade `c` a uma cidade `j` é o menor número de estradas que preciso percorrer para ir de `c` a `j`. (A distância de `c` a `j` é, em geral, diferente da distância de `j` a `c`.) Nosso problema: dada uma cidade `c`,

determinar a distância de `c` a cada uma das demais cidades.

As distâncias serão armazenadas em um vetor `d`: a distância de `c` a `j` será `d[j]`. Que fazer se for impossível chegar de `c` a `j`? Poderíamos dizer nesse caso que `d[j]` é infinito. Mas é mais limpo e prático dizer que `d[j]` vale 6, pois nenhuma distância "real" pode ser maior que 5. Se tomarmos `c` igual a 3 no exemplo acima, teremos

i	0	1	2	3	4	5
d[i]	2	3	1	0	1	6

Eis a ideia de um algoritmo que usa o conceito de fila para resolver nosso problema:

- digamos que uma cidade é *ativa* se já foi visitada mas as estradas que começam nela ainda não foram exploradas;

- mantenha uma fila das cidades ativas;
- em cada iteração, remova da fila uma cidade `i` e insira na fila todas as cidades vizinhas a `i` que ainda não foram visitadas.

```
// Recebe uma matriz A que representa as interligações
// entre cidades 0,1,..,5: há uma estrada (de mão única)
// de i a j se e só se A[i][j] == 1.  Devolve um vetor
// d tal que d[i] é a distância de c a i para cada i.

int *distancias( int A[][6], int c) {
   int *d, j;
   int fila[6], ini, fim;

   d = mallocc( 6 * sizeof (int));
   for (j = 0; j < 6; ++j)  d[j] = 6;
   d[c] = 0;
   ini = 0; fim = 1; fila[0] = c;  // c entra na fila

   while (ini != fim) {
      int i, di;
      i = fila[ini++];             // i sai da fila
      di = d[i];
      for (j = 0; j < 6; ++j)
         if (A[i][j] == 1 && d[j] >= 6) {
            d[j] = di + 1;
            fila[fim++] = j;       // j entra na fila
         }
   }
   return d;
}
```

Para compreender o algoritmo (e provar que ele está correto), observe que as seguintes propriedades invariantes valem no início de cada iteração:

1. para cada cidade `x`, se `d[x]` < 6 então `d[x]` é a distância de `c` a `x`;
2. a sequência de números `d[fila[0]]`, ... , `d[fila[fim-1]]` é crescente;
3. se `d[fila[ini]]` vale `k` então `d[fila[fim-1]]` ≤ `k+1`;
4. para cada índice `h` em `0..ini-1`, toda estrada que começa em `fila[h]` termina em algum elemento de `fila[0..fim-1]`.

Exercícios 2

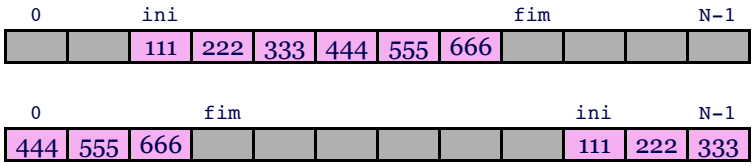
1. Reescreva o código da função `distancias` sem os "`++`". [Solução]
2. Quem garante que a instrução "`fila[fim++] = j`" não provoca o transbordamento da fila? Em outras palavras, quem garante que o espaço alocado para o vetor `fila` é suficiente?
3. Prove que a função `distancias` de fato calcula as distâncias corretas. Para isso, prove antes as propriedades invariantes enunciadas acima.
4. Reescreva a função `distancias` para um número arbitrário de cidades. Faça uma versão que devolva a distância entre duas cidades dadas.
5. Imagine um tabuleiro quadriculado com $m \times n$ casas dispostas em m linhas e n colunas. Algumas casas estão livres e outras estão bloqueadas. As casas livres são marcadas com "-" e as bloqueadas com "#". Há um robô na casa (1,1), que é livre. O robô só pode andar de uma casa livre para outra. Em cada passo, só pode andar para a casa que está "ao norte", "a leste", "ao sul" ou "a oeste". Ajude o robô a encontrar a saída, que está na posição (m,n). (Sugestão: Faça uma moldura de casas bloqueadas.)

Implementação circular

No exemplo das distâncias, é fácil ver que o vetor que abriga a fila não precisa ter mais componentes que o

número total de cidades, pois cada cidade entra na fila no máximo uma vez. Em geral, entretanto, é difícil prever o espaço necessário para abrigar a fila. Nesses casos, é mais inteligente implementar a fila de maneira "circular", como mostraremos a seguir.

Suponha que os elementos da fila estão dispostos no vetor `fila[0..N-1]` de uma das seguintes maneiras: `fila[ini..fim-1]` ou `fila[ini..N-1] fila[0..fim-1]`.



Teremos sempre `0 ≤ ini < N` e `0 ≤ fim < N`, mas não podemos supor que `ini ≤ fim`. A fila está

- *vazia* se `fim == ini` e
- *cheia* se `fim+1 == ini` ou `fim+1 == N` e `ini == 0`.

Resumindo, a fila está cheia se `(fim+1) % N == ini`. A posição `fim` ficará sempre desocupada, para que possamos distinguir uma fila cheia de uma vazia. Para remover um elemento da fila basta fazer

```
x = fila[ini++];
if (ini == N) ini = 0;
```

É claro que isso só deve ser feito se você souber que a fila não está vazia. Para inserir um elemento `y` na fila faça

```
if (fim + 1 == ini || fim + 1 == N && ini == 0) {
   printf( "\nSocorro! Fila vai transbordar!\n");
   exit( EXIT_FAILURE);
}
fila[fim++] = y;
if (fim == N) fim = 0;
```

O código começa por verificar se a fila está cheia; se estiver, podemos pedir socorro e abortar o programa ou podemos redimensionar a fila.

Exercícios 3

1. Resolva os seguintes problemas a respeito da implementação circular de uma fila. (Lembre-se de que uma fila é um pacote com três objetos: um vetor e dois índices.) (a) Escreva uma função que remova um elemento da fila. Quais são os parâmetros da função? (b) Escreva uma função que insira um número na fila. (c) Escreva uma função que devolva o comprimento (ou seja, o número de elementos) da fila.
2. REDIMENSIONAMENTO. Se a fila ficar cheia, é uma boa ideia alocar um novo vetor e transferir a fila para esse novo vetor. Escreva uma função que faça esse *redimensionamento*. É uma boa ideia fazer com que o novo vetor tenha o dobro do tamanho do original.
3. Uma *fila dupla* (= *deque*) permite saída e entrada em qualquer das duas extremidades da fila. Implemente uma fila dupla e programe as funções de manipulação da estrutura.

Fila implementada em uma lista encadeada

Como administrar uma fila armazenada em uma lista encadeada? Digamos que as células da lista são do tipo `celula`:

```
typedef struct cel {
   int         conteudo;
   struct cel *prox;
} celula;
```

É preciso tomar algumas decisões de projeto sobre como a fila vai morar na lista. Vamos supor que nossa lista encadeada é *circular*: a última célula aponta para a primeira. Vamos supor também que a lista tem uma *célula-cabeça*; essa célula não será removida nem mesmo se a fila ficar vazia. O primeiro elemento da fila ficará na *segunda* célula e o último elemento ficará na célula *anterior* à célula-cabeça.

Um ponteiro `fi` que guardará o endereço da célula-cabeça. A fila está *vazia* se `fi->prox == fi`. Uma fila vazia pode ser criada e inicializada assim:

```
celula *fi;
fi = mallocc( sizeof (celula));
fi->prox = fi;
```

Podemos agora definir as funções de manipulação da fila. A remoção é fácil:

```
// Remove um elemento da fila fi e devolve
// o elemento removido. Supõe que a fila
// não está vazia.

int remove( celula *fi) {
   int x;
   celula *p;
   p = fi->prox;  // p aponta primeiro da fila
   x = p->conteudo;
   fi->prox = p->prox;
   free( p);
   return x;
}
```

A inserção exige um truque sujo: armazenar o novo elemento na célula-cabeça original:

```
// Insere um novo elemento com conteudo y
// na fila fi. Devolve o endereço da
// cabeça da fila resultante.

celula *insere( int y, celula *fi) {
   celula *nova;
   nova = mallocc( sizeof (celula));
   nova->prox = fi->prox;
   fi->prox = nova;
   fi->conteudo = y;
   return nova;
}
```

Exercícios 4

1. Implemente uma fila em uma lista encadeada circular *sem* célula-cabeça. Basta manter o endereço `fim` da última célula; a primeira célula será apontada por `fim->prox`. Se a lista encadeada estiver vazia então `fim == NULL`.
2. Implemente uma fila em uma lista encadeada simples (não circular) com célula-cabeça. Será preciso manter o endereço `ini` da célula-cabeça e um ponteiro `fim` para a última célula.
3. Implemente uma fila em uma lista encadeada simples sem célula-cabeça. Será preciso manter um ponteiro `ini` para a primeira célula e um ponteiro `fim` para a última.
4. Implemente uma fila em uma lista *duplamente* encadeada *sem* célula-cabeça. Use um ponteiro `ini` para a primeira célula e um ponteiro `fim` para a última.

Veja o verbete *Queue* na Wikipedia

Last modified: Mon Apr 13 07:12:53 BRT 2015
http://www.ime.usp.br/~pf/algoritmos/
Paulo Feofiloff
DCC-IME-USP